Very large database applications of the Connection Machine system

by DAVID WALTZ, CRAIG STANFILL, STEPHEN SMITH, and ROBERT THAU
Thinking Machines Corporation
Cambridge, Massachusetts

DR87-3

ABSTRACT

The architecture of the Connection Machine™ system is particularly appropriate for large database applications. The Connection Machine system consists of 65,536 processors, each with its own memory, coupled by a high speed communications network. In large database applications, individual data elements are stored in separate processors and are operated on simultaneously. This paper examines three types of applications of this technology. The first, which will be examined in the greatest detail, is the use of the Connection Machine System for document retrieval. The second application is parsing large free text databases and preparing them for searching. The third topic is the application of the Connection Machine to associative memory or content addressable memory tasks. This ability has been put to use in a method called "memory-based reasoning" which can produce expert system-like behavior from a database of records of earlier decisions.

INTRODUCTION

The rapid growth of on-line databases is a great challenge to information processing technology. First, databases are growing quickly in size: databases with tens of gigabytes are now quite common, and databases with hundreds of gigabytes are by no means unknown. Soon, with advances in optical disk technology, we will have to deal with databases having terrabytes or even tens of terrabytes of data.

This growth in database size creates two problems. The first is obvious: when a database doubles in size, the amount of storage it requires doubles, and the amount of compute power necessary to process it doubles. The second is perhaps less obvious: as a database grows, it becomes more and more difficult to locate and manipulate the information it contains.

For text databases, this second problem manifests itself as deteriorating search quality: as a database contains more and more information, finding the right information becomes increasingly difficult, so that the system is likely to miss desired information or to deluge the user with unwanted data. This problem can be (at least partly) overcome by the use of more sophisticated retrieval algorithms such as relevance feedback (see below). These algorithms are, however, computationally more expensive than the algorithms generally used for document retrieval today.

Databases containing numerical and symbolic data also require more intensive processing as they grow in size. In the current paradigm, a database might be abstracted to a much smaller set of statistical characteristics (e.g., "The average salary of the CEOs of Fortune 500 corporations"). However, in abstracting a database to a few statistical parameters information is inevitably lost. We believe a good solution to this problem is to search the database for precedents, a technique called "Memory Based Reasoning" (see below). In this method, decisions are made by searching a database for episodes similar to the problem facing the user, then basing conclusions on the data so located. Again, problems associated with large databases may be attacked, but the algorithms are computationally intensive.

Thus, it is desirable that computational power increase faster than the size of the database. Unfortunately, just the opposite is happening, as the performance of serial computers has reached a plateau while databases continue to get bigger and bigger. A solution to this problem is the use of a new generation of parallel computers, such as the Connection Machine™ System, which have much greater computational power than conventional (serial) machines. The remainder of this paper will consider some specific applications of this new machine to problems associated with large databases.

DOCUMENT RETRIEVAL

Document retrieval has traditionally been implemented as Boolean search on an inverted file. The main difficulties of Boolean search are that: 1) users require considerable training in the use of a query language, and 2) users generally alternate between being overwhelmed by too many documents if one uses a too general search pattern, or too few documents if one is more restrictive.[1]

We have built an easy-to-use document retrieval system that allows simultaneous searches of very large databases by a large number of users.[2] The system mixes AI ideas with methods from information science. Its basis is a weighted associative memory algorithm. In contrast to a Boolean search system, a naive user can be trained to use our system in a few minutes. The system operates very rapidly, and has high precision and recall.

Using the algorithm described here, a single Connection Machine system allows a 6 GByte free text database to be searched and browsed rapidly and conveniently by over 2000 simultaneous users. Other algorithms (not yet implemented) that are dependent on a high speed multiple disk mass storage unit will allow much larger databases to be searched.

Relevance Feedback

From the user's point of view, the search process on the Connection Machine document retrieval system has two distinct phases. In the first phase, the user types a list of a few keywords, for example, "Iran Contra arms deal." The retrieval system returns a list of documents, ordered according to how many of the keywords they contain, and how important each keyword is (the rarer the word, the more important). In the second phase, the user browses through these documents and finds one or more that bear on the topic of interest. As relevant documents are located, the user may command the system to search for related documents by performing a full-text to full-text comparison between the documents he has already found and every document in the database. This is done by automatically extracting the words from the text of the known relevant documents and rating the documents in the database according to how many of those words they contain.

This method, termed "relevance feedback," has been known since the 1960s to yield high quality searches, but to the best of our knowledge, it has never been used on a large database because of its extremely high computational requirements.

Implementation

Relevance feedback generates queries containing hundreds or thousands of terms, where each term consists of a word and the weight assigned to that word. Each processor in the system is assigned a 25 word segment from a single document. To execute a query, a serial front end processor broadcasts the word-weight pairs to the Connection Machine System. As this is done, each processor tests its segment for the presence of each word. When a word is found, the processor accumulates the word's weight to form a score for that document. After the complete query has been broadcast, the results are sorted in order of decreasing score, and the pointers to the 20–100 documents with the highest scores are sent to the user.

Documents are represented using a method of *surrogate coding* described by Stanfill and Kahle.[2] In this method, groups of 25 words are collected and used to set bits in a 1024 bit vector. Bits are set by applying n hash code functions to each term (n typically = 10). Thus, to represent a document with 75 terms, about 250 bits in each of three vectors of 1024 bits each would be set. Occasionally, more than one hash function may set the same bit. This is handled by superimposing the results. The vectors for each document are then stored in a contiguous group of processors.

In order to search for a document, the same hash codes that were used to construct the bit vectors are applied to each of the terms in the search pattern, one at a time, and the positions of the hash code bits are broadcast to all the processors. Each processor checks to see whether it has all 10 bits set for a given term and if so, it adds a score for that term to the total score for the document stored in its document mailbox. This algorithm is probabilistic: there is a small chance that a given term will hash into 10 locations which were set by other terms, causing the system to interpret it as a term occurring in a document when in fact it has not. The probability of this happening is dependent on the number of hash functions applied to each term, the size of the vector, and the number of terms in the overall database. The probability of a false hit can be made arbitrarily small by applying more hash functions or increasing the length of a bit vector. With the parameters we have been using, the probability of a false hit is about 1/1,000,000. Additionally, since each query in a relevance search contains an average of 75 terms, one or two false hits cannot make much of a difference in the overall relevance calculation.

Performance

The performance figures contained in this section assume that a Connection Machine system with 65,536 processors and 2 GBytes of fast memory. Furthermore, we assume the data preparation algorithms explained below have been applied, yielding an overall data compression rate of 3.3:1. Finally, we assume a user spends two minutes browsing between searches.

The bit vectors are much smaller than the memories of the individual processing elements. This allows us to use the Connection Machine's "virtual processor" mechanism to increase the amount of data stored in the system. To do this, each processor's memory is segmented, and the processors sequentially perform any computation on each memory segment. To the programmer or user, it appears that there are n times as many processors, each with $1/n$ as much memory as a real processor, each operating at $1/n$ the speed.[3,4]

The 2GBytes of memory in the Connection Machine System allows us to store surrogates built from 7 Gigabytes of raw text. Each processor will then simulate 64 virtual processors, and run at 1/64th the speed of a physical processor. Each surrogate may be tested for the presence of a word in 2 microseconds, and a score may be added in 6 microseconds. Allowing for the virtual processor ratio, this allows us to execute a single term in 512 microseconds. If we assume an average query generated by relevance feedback contains 75 terms, we may then perform a complete search in 38 milliseconds. This would allow our system to support over 2000 users.

FORMATTING A DATABASE

As the size of databases and the rate at which data is fed into a system increases, the problem of scanning raw data, indexing it, and adding it to the database becomes ever more difficult. We are attacking this problem by building a set of natural language and text processing tools running on the Connection Machine System.[5,6,7]

Formatting a database proceeds in several phases, gradually grouping the individual characters of the raw file into words, phrases, paragraphs, and documents. First, we use a regular expression-based lexical analysis system to break the input stream into tokens, such as words and punctuation marks. Second, several dictionaries are used to differentiate important from unimportant words and to group words into known phrases. Finally, we take groups of words, put them into surrogate tables, and write the results out to disk. We are also working on generalized parsing algorithms, which may be applied to identifying specialized forms of noun phrases (e.g., names of people, places, dates), as well as to full syntactic parsing. These parsing tools have yet to be integrated into the full system.

Lexical Analysis

The first step in processing raw text (e.g., a newswire transcript) is parsing it into meaningful units. This must be done on two levels. The words in the text must be found (and distinguished from, say, embedded formatting directives). Just as important, the incoming stream of raw text must be split into separate documents, with paragraph delimiters, identified headlines, and dates. All of this is done by a regular-expression based lexing phase, which runs the regular expressions by using precompiled finite state automata (FSAs).

The lexer is driven off an action list, which contains pointers to FSAs, and indications of what to do with the matches. For example, paragraph delimiters are handled by having the action list run an FSA which finds paragraph boundaries and inserts a special delimiting sequence. This regular expression is specific to the source of text: it might look for indented lines, blank lines, or embedded formatting directives.

The bulk of the work is done by two special directives. One of them marks all substrings of the input matching a regular expression as words. The other splits undelimited text into documents while extracting information such as the headline and date.

There are two algorithms for running FSAs on a Connection Machine System. The most direct is to put a copy of the FSA in each processor, and stream the text by them. Spurious matches like the "have" in "behave" are suppressed, by using the first match that ends at a given point. This is fine for FSAs that have only to match short strings (e.g., words), but it is excessively time consuming when the FSA needs to match a large amount of text (e.g., the headline of a newswire article). In these circumstances, the log-time lexing algorithm of Hillis' and Steel's "Data Parallel Algorithms"[8] is more appropriate.

Dictionaries

The text compression algorithm uses word frequency dictionaries. These dictionaries consist of words paired with a count of how many times they occurred in a corpus of known size. One dictionary entry is stored per processor.

To construct such a dictionary from a textual database requires four distinct steps: 1) load the Connection Machine with text from the given database, 2) accumulate the characters such that there is one word per processor, 3) produce dictionary entries for these new words, and 4) merge these new entries with the existing dictionary. This last step is accomplished by sorting the entries alphabetically, grouping entries with the same word, and summing the count fields for each group.

Look-up word frequencies in these dictionaries proceeds similarly to the dictionary building phase described above: 1) a "dummy" entry is created for each word to be looked up. This dummy contains a pointer back to the processor requesting the word's definition. 2) the dummies are sorted with the real dictionary entries so that dummy entries always follow the real entries for a word. 3) entries for the same word (both real and dummy) are grouped together. 4) definitions are copied from real to dummy entries within a group. 5) the definitions are sent back to the requesting processor via the dummy's backpointer.

Frequency Based Indexing

The text compression algorithm being used in the current Document Retrieval System is called "Frequency Based Indexing." This algorithm extracts content bearing terms based on the number of times they occur throughout some large database. Words with high frequencies, such as "the," "of," and "to" are dropped. Words with low frequencies, such as "parallel," "computer," and "algorithm" are retained. Proper nouns like "Iran" and "Reagan" are always retained.

For words which are neither rare enough to be dropped outright nor frequent enough to be automatically retained, we use a word-pair based method. For example, the word "special" is too common to be retained by the frequency considerations alone. However, when we build a dictionary of word-pair frequencies, we find the two word phrase "special prosecutor" is more common than the frequencies of "special" and "prosecutor" alone would suggest. Thus, the word "special" is retained, and marked as part of a phrase.

Building Surrogates

The final step in formatting a database is building the surrogate tables. First, each processor computes the ten hash codes for a single word. Next, 1024 bits are zeroed out in each processor as a partial hash table. The bits that the ten hash codes pointed to are then set. Finally, the 25 tables corresponding to a document segment are ORed together to form the final surrogate.

Parsing

At the moment, the system only discriminates between words on the basis of frequency. An improvement might be to detect specific types of multi-word phrases. A quick and effective way of doing this is to retrieve part-of-speech information from the dictionary and use an FSA to recognize phrases. Preliminary experiments using a dictionary of words likely to appear in names of corporations indicate that this approach can be used to find names of companies and such, with some success. Experiments have also been done with full text parsing.[9]

MEMORY-BASED REASONING

Memory-Based Reasoning (MBR) is a new paradigm for AI in which an associative memory using a best-match algorithm takes the place of rules. It is particularly well-suited to massively parallel computers such as the Connection Machine System. Memory-Based Reasoning places memory at the foundation of intelligence, rather than at the periphery. Memories of specific events are used directly to make decisions, rather than indirectly (as in systems which use experience to infer rules). In its purest form, memory-based reasoning uses the global nearest match computation to find the items in memory most similar to a current situation and then uses the actions associated with these items to deal with the current situation. In essence, reasoning is reduced to perception: the current situation is observed, it reminds the system of something it has seen before, and an immediate reaction is forthcoming without further analysis.

We can contrast memory-based reasoning in this extreme form with models based on heuristic search. In heuristic search, solutions are generated rather than looked up. To give a concrete example, in solving a medical reasoning problem, a memory-based reasoning system finds a patient or patients most similar to the current patient by a global nearest match operation, and uses the diagnosis, treatment, and outcome to find a diagnosis and treatment, and to predict an outcome for the current patient. A rule-based forward chaining system takes the patient's symptoms and applies rules one after another until it arrives at a diagnosis.

The advantages of memory-based reasoning are: 1) it is much easier to generate examples than to generate rules, so the knowledge acquisition for a memory-based reasoning system is much simpler; 2) memory-based reasoning systems inherently have a mechanism for judging confidence in an answer—if there is a very close match in memory to the current situation, one can be quite confident of the outcome. If the nearest match is far away, the system can note that its results are uncertain, "it can know that it doesn't know"; 3) memory-based reasoning systems can scale well to very large problems, and a single system can handle simultaneously a number of different kinds of problems.

Significantly, parallel hardware reverses the relative efficiency of memory-based reasoning and rule-based reasoning. On serial hardware, the best-match operation is very expensive because every data item must be considered in turn, while rule-triggering is relatively cheap due to the existence of algorithms (e.g., Rete networks) that allow a database of rules to be efficiently searched. On parallel hardware, the best-match algorithm takes constant time, while rule-invocation takes time proportional to the number of rules which must be chained to obtain an answer.

The remainder of this section will discuss work to date on memory-based reasoning, including experiments with pure MBR and MBR augmented with a generalization mechanism. We will then present our plans for further development of the paradigm.

Work to Date

In this section we will discuss memory-based reasoning applied to the classification problem. Given a database of objects, each object belonging to one of a set of mutually exclusive classes, we classify new objects by finding the best match in the existing database and looking at its class.

As reported in Stanfill's and Waltz' "Toward Memory-Based Reasoning,"[10] we implemented a memory-based reasoning "shell." This shell computes the nearest match to an input pattern (which may be incomplete) using a set of weighting and distance measures, the net effect of which is to find the distance from every individual example in memory to the current example to be classified.

The shell assumes a relational database-like format for examples. More formally, a database is a set of records. Each record has a fixed set of fields. The field specifying the class of a record is the goal field, and the other fields are predictor fields. Novel records which are to be classified are target records.

The computation of similarity is fairly complex. Field weights are computed by judging how tightly a particular predictor field constrains values of the goal field. The distance between two records is then computed by summing the weights for all predictor fields for which they have different values. For example, if a patient reports having a sore throat, this constrains the range of diseases he/she might be suffering from to a relatively small range (e.g., a viral infection, a strep infection, smoking). Thus, if a patient reported a sore throat, all records in the database which did not include a sore throat would receive a large distance measure. On the other hand, having a low fever places relatively few constraints on the possible maladies, so it would receive a small weight.

More recently, we have added a generalization algorithm to the memory-based reasoning shell. This algorithm searches for patterns in the database (e.g., "a high fever accompanied by a sore throat indicates a strep infection"), and remembers which records obey them. These stored patterns are then used to augment the best-match process: if a target record matches a stored pattern, the data records used to generate the pattern will have their similarity-measures boosted. The primary benefit of this generalization mechanism is to significantly reduce the system's sensitivity to noise. In all cases, it produces a significant improvement in the quality of MBR's decisions.

An Experiment

Memory-Based Reasoning was applied to the problem of pronouncing English words. The formulation of the task is deliberately similar to Sejnowski and Rosenberg's NETtalk system.[11] In this case, the database is a dictionary.[12] Each record in the database consists of a seven-letter window in a word (a letter, the three previous letters, and the next three letters); a three-phoneme window in the pronunciation (the phoneme plus the two preceding phonemes); and the stress of the letter (primary stress, secondary stress). For example, the word "file," which has pronunciation "fAL-" and stress pattern "1- -," would yield the following four records:

*	*	*	f	i	l	e	*	*	f	+
	*	f	i	l	e	*	*	f	A	1
	f	i	l	e	*	*	f	A	1	-
f	i	l	e	*	*	*	A	l	-	-

The 20,000 words in the dictionary thus yield 146,951 records.

It must be noted that perfect performance on the pronunciation task, as outlined above, is fundamentally impossible. First, many English words are borrowed from other languages, often retaining their original pronunciations. Thus, any system which pronounced "montage" correctly would almost certainly mispronounce "frontage." Second, the stress patterns of English words often depend on their part of speech. In some cases, a word will even have two acceptable pronunciations, depending on whether it is used as a noun or a verb ("to object" versus "an object").

In spite of the difficulty of the pronunciation task, Memory-Based Reasoning does quite well. Given the three preceding letters, the three succeeding letters, the two preceding phonemes, and the stress, MBR produces the correct phoneme 92% of the time. If we omit the previous two phonemes and the stress, MBR gets the correct phoneme 87% of the time. With a database of 128K records running on a 32K processor Connection Machine System, each classification is accomplished in 30 milliseconds.

Evaluation

The two MBR algorithms described above (pure MBR and MBR with generalization-learning) were evaluated according to sensitivity to database size, distraction, and noise. The results of these experiments are discussed more fully in Stanfill's "Memory-Based Reasoning Applied to English Pronunciation."[13]

Database size

Both algorithms exhibit graceful degradation as the size of the database shrinks from 128K down to 4K. The generalization algorithm is always slightly better. With 4K records (approximately 700 words), 78 percent of phonemes were correct.

Distraction

In an effort to distract the algorithms, between one and seven fields containing random values were added to each record in the database. These had no effect on either algorithm.

Noise

Two types of noise were considered. First, between 10% and 100% noise was added to the predictor fields.* Neither algorithm was significantly affected until noise exceeded 90%, at which point performance collapsed. Second, between 10% and 100% noise was added to the goal fields. For pure MBR, performance fell about linearly with added noise. For MBR with generalization, performance degraded more slowly until the noise level exceeded 60%.

Prospects for Memory-Based Reasoning

Work, so far, has concentrated on the application of pure memory-based reasoning to "flat" relational databases, with the representation fixed by the system builder. The next stages will be to relax some of these restrictions. Part of this work has already been started, with the addition of generalization to MBR. We also plan to allow MBR to modify its representations, as well as to allow for a greater flexibility in their form (e.g., allowing networks and hierarchies).

In the long run, we believe that memory-based reasoning will provide a unifying paradigm for Artificial Intelligence. Most aspects of intelligence, we believe, can be expressed as operations on or augmentations to memory. This includes perception, attention, generalization, learning, and deduction. Indeed, aspects of some of these phenomena appear as emergent behavior of the simple MBR model presented above.

* For 10% noise, 10% of the predictor-fields in the database would receive a random value.

REFERENCES

1. Blair, D. C., and M. E. Maron. "An Evaluation of Retrieval Effectiveness for a Full-Text Document-Retrieval System." *Communications of the ACM,* 28 (1985) 3, pp. 285–299.
2. Stanfill, C., and B. Kahle. "Parallel Free Text Search on the Connection Machine System." *Communications of the ACM,* 29 (1986) 12, pp. 1229–1239.
3. Thinking Machines Corporation, *Introduction to Data Level Parallelism.* Cambridge, MA, April, 1986.
4. Hillis, D. *The Connection Machine.* Cambridge, MA: MIT Press, 1985.
5. Sabot, G., "Bulk Processing of Text on a Massively Parallel Computer." *Proceedings 24th Annual Meeting of the Association for Computational Linguistics,* New York, June 10–13, 1986, pp. 128–135.
6. Waltz, D. L. "Applications of the Connection Machine." *IEEE Computer,* 20 (1987) 1, pp. 85–97.
7. Smith, S. "Extracting Content Bearing Terms in Parallel on the Connection Machine," Thinking Machines Corporation Technical Paper DR87-1, 1987.
8. Hillis, D., and G. Steel. "Data Parallel Algorithms." *Communications of the ACM,* 29 (1986) 12, pp. 1170–1183.
9. Thau, R., and S. Ferguson. "Context Free Parsing on the Connection Machine System." Thinking Machines Corporation Technical Paper NL87-1, 1987.
10. Stanfill, C., and D. L. Waltz. "Toward Memory-Based Reasoning." *Communications of the ACM,* 29 (1986) 12, pp. 1213–1228.
11. Sejnowski, T. J. and C. R. Rosenberg. "NETtalk: A Parallel Network that Learns to Read Aloud." The Johns Hopkins University Electrical Engineering and Computer Science Technical Report JHU/EECS-86.
12. *Merriam Webster's Pocket Dictionary,* 1974.
13. Stanfill, C. "Memory-Based Reasoning Applied to English Pronunciation." *Proceedings, Sixth National Conference on Artificial Intelligence (AAAI-87),* Seattle, Washington, July 13–17, 1987.